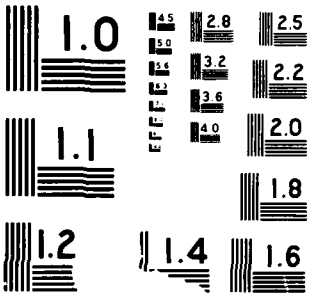
AD-A193 648 COMBINED AND-OR PARALLEL EXECUTION OF LOGIC PROGRAMS (U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE G GUPTA ET AL. MAR 88 TR88-012 1/1

UNCLASSIFIED N00014-86-K-0680 F/G 12/5 NL

[redacted]

END
DATE FILMED
7 88


1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

(4)

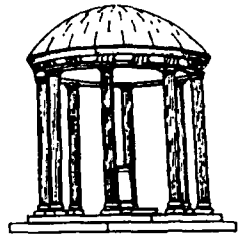
Combined And-Or Parallel Execution of Logic Programs

TR88-012

March 1988

Gopal Gupta and Bharat Jayaraman

N00014-86-K-0680



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Combined And-Or Parallel Execution of Logic Programming Languages†

Gopal Gupta
Bharat Jayaraman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract

A number of approaches have recently been proposed for the parallel execution of logic programming languages, but most of them deal with either or-parallelism or and-parallelism but not both. This paper describes a high-level design for efficiently supporting both and-parallelism and or-parallelism. Our approach is based on the 'binding arrays' method for or-parallelism and the 'RAP' method for and-parallelism. Extensions to the binding-arrays method are proposed in order to achieve constant access-time to variables in the presence of and-parallelism. The RAP (Restricted And-Parallelism) method becomes simplified because backtracking is unnecessary in the presence of or-parallelism. Our approach has the added effect of eliminating redundant computations when goals exhibit both and- and or-parallelism. The paper first briefly describes the basic issues in pure and-parallelism and or-parallelism, states desirable criteria for their implementation (with respect to variable access, task creation and switching), and then describes the combined and-or implementation.

† This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

1. Introduction

Logic programming [K74] has attracted great interest recently because of its applicability in symbolic computation, parsing, intelligent databases, and expert systems. From the standpoint of implementation, an important characteristic of logic programming languages is that they are amenable to highly parallel execution, because they disallow destructive assignments and explicit sequencing. Conery identifies two important forms of parallelism in logic programming languages: or-parallelism and and-parallelism [CK81]. Or-parallelism arises when a goal can be matched with multiple rules and these multiple paths can be pursued in parallel. And-parallelism arises when multiple goals can be executed in parallel. However, realizing or- and and-parallelism in an actual implementation poses significant challenges: to realize or-parallelism, we must efficiently represent and access the multiple bindings for variables [W87]; and to realize and-parallelism, we must avoid a time-consuming dependency analysis to ensure that goals are independent of one another [D85]. This paper is concerned with strategies and techniques for the parallel implementation of logic languages, with the goal of efficiently realizing both and-parallelism and or-parallelism.

A number of approaches have already been proposed for the parallel execution of logic programming languages, but the bulk of current research has dealt with either or-parallelism [BL86, CH86, C87, DLO87, HCH87, JG86, L84, SW87, TL87, W84, W87] or and-parallelism [CK83, D84, D87, H86, HN86, LKL86]. Our experience with practical logic programs suggests that both forms of parallelism do arise naturally, although most programs tend to exhibit predominately one form of parallelism. In this paper we devise a framework for the combined and-or parallel execution of logic languages. Our approach builds upon the best known techniques for exploiting or-parallelism and and-parallelism: the 'binding arrays' method for or-parallelism [W84, W87] and the 'RAP' method for and-parallelism [D84, HN86]. Essentially, our approach extends the pure or-parallel implementation by providing for the sharing of results across (independent) and-parallel computations. The approach has the following properties: (i) variables are accessed in constant-time, (ii) task creation is also constant-time, (iii) redundant computation is avoided when goals exhibit both and- and or-parallelism, (iv) and-parallel goals need not be

restarted upon backtracking as in pure and-parallel systems (because there is no backtracking in our approach), and (v) early pruning of failing and-parallel goals is possible (as in intelligent backtracking). Our approach has one potential disadvantage: task-switching time is not constant. To minimize this overhead, we propose to adopt a fairly high task-granularity. We assume a shared-memory MIMD model in which all processors can access all memories in constant-time.

The rest of this paper is organized as follows: section 2 describes the major issues in or-parallelism, states desirable criteria for an or-parallel implementation, and then describes the binding-arrays method; section 3 similarly describes the issues and criteria for and-parallelism, and provides a brief description of the restricted and-parallelism method; section 4 describes the combined and-or parallel implementation; and finally, section 5 presents conclusions and further comments on related work.

2. Or Parallelism

Or-parallelism manifests itself whenever there is a non-deterministic search for solutions. In logic programs, or-parallelism arises when multiple clause heads unify with a goal. The subgoals arising from these multiple matches can be executed in parallel. The following very simple example illustrates the basic idea.

```
father(adam, cain).
father(adam, abel).
mother(eve, cain).
mother(eve, abel).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

Given the above logic program, the goal

```
?  parent(P, C)
```

can match both clause heads for `parent`, and the subgoals `father(P, C)` and `mother(P, C)` can be executed in or-parallel fashion. Typically, or-parallel execution is initiated by creating a separate *task* corresponding to each successful match, and executing the tasks in parallel. Thus, all parent-child pairs can be computed simultaneously. Figure 1 shows the goal tree for the above goal; we refer to this

tree as the or-parallel tree. Each node of the tree records the local variables of the clause that unified with the parent goal of this node.

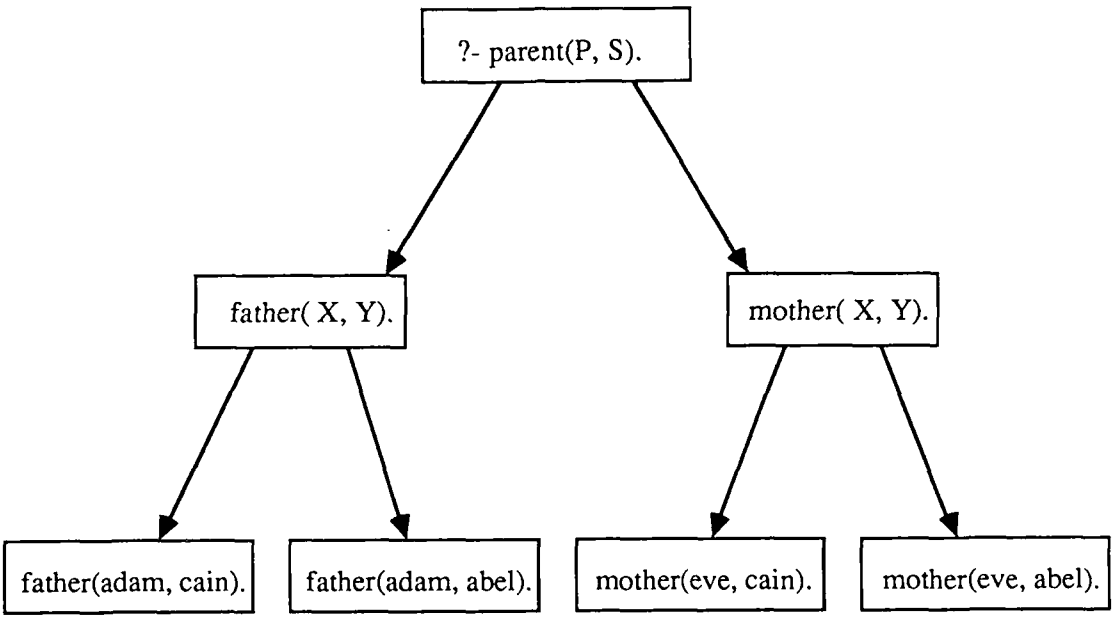


Figure 1. A Pure Or-parallel Tree.

2.1. Criteria for Or-parallel Implementations

The above example illustrates one of the basic requirements of an or-parallel implementation: it must be able to represent multiple bindings for certain variables, e.g., P and C. This is in sharp contrast with sequential implementations of Prolog, such as described in [WPP77], where multiple solutions are explored one at a time, using a trail stack to record variables like `P` and `C` that need to be reset upon backtracking. In general, all unbound variables appearing in the argument terms of a goal could potentially obtain multiple bindings during or-parallel execution. D.H.D. Warren refers to such variables as *conditional variables* [W87].

In addition to representing multiple bindings, an efficient implementation must ensure that the *access time* to such conditional variables and the *task-creation time* needed is not prohibitive; ideally, they should be a constant independent of the size

of the goal tree and independent of the size of the arguments of a goal. Because the number of bindings for a conditional variable cannot be predicted in advance (because it depends on the number of solutions to the goal), nor is it immediately obvious which of the multiple bindings is applicable to some descendent task, the representation of the multiple bindings must cater to constant access time. Task creation should not, for example, traverse arguments of a goal to determine which variables are unbound; nor should it copy bindings for all variables on the path from the root of the goal tree to the node creating the new task. Although the different or-parallel paths are logically independent, they can (and should) share the bindings for all bound variables in the path from the root to any common ancestor of two or-parallel nodes.

A further requirement on or-parallel implementations arises from the finite nature of the underlying parallel machine. Because it is very likely that the number of or-parallel tasks will exceed the number of available processors—a valid assumption for commercially available parallel machines—the *task switching time* should also not be prohibitive, and ideally a constant independent of the goal tree. We make the assumption that the underlying machine provides constant access-time to all memory locations—an assumption that is valid only for shared-memory multiprocessors. We can thus sum up the criteria for an ideal or-parallel implementation as follows:

1. constant access time to all variables;
2. constant task-creation time; and
3. constant task-switch time.

Other desirable characteristics of an ideal or-parallel implementation are that it should execute as efficiently as a sequential implementation in case only one processor is available. Also, it should be amenable to optimizations that apply to sequential implementations, such as last-call optimization and environment trimming.

2.2. The Binding Arrays method

A number of approaches to or-parallel implementation of logic programs have re-

cently been proposed:

1. Hashing windows [B84];
2. OR-parallel token machine [CH86];
3. Variable importation [L84];
4. Time-stamping [TL87];
5. Version-vectors [HCH87];
6. Environment-closing [C87];
7. Favored-bindings [DLO87, SW87];
8. Binding Arrays [W84, W87].

We do not attempt a description of all these approches in this report. Reference [W87] provides a comparison of some of these approaches. In our opinion, no method achieves all the criteria mentioned earlier; D.H.D. Warren [W87] argues that there is no clearly superior technique either. Below we describe briefly the binding-arrays method of D.S. Warren [W84], as enhanced by D.H.D. Warren [W87], because it performs as well as the best methods, and has the further property that it can be adapted for a combined and-or parallel implementation, to be described in section 4.

In this method, each node of the goal tree contains the *local variables* of the clause that successfully unified with the parent subgoal, and also a *binding list.* When a node binds a conditional variable in one of its ancestor nodes, it stores an $<a, v>$ pair in its binding list, where a is the address of the conditional variable in the ancestor node, and v is a pointer to the assigned value. Thus, all variables whose addresses would have been *trailed* in a sequential implementation end up on the binding list. The binding list is needed to perform task-switching, explained further below. To make access to conditional variables a constant-time operation, each processor has an array called the *binding array*, which is initially empty, but gets updated dynamically as explained below.

Task Creation. All local variables of a node that are unbound at the time it creates one or more or-parallel subgoals are assigned consecutive indices in the binding array. A counter is maintained with each node for this purpose, and is initially zero in the root node. The new value of the counter is then copied into the

nodes for each of the subgoals created—to be used when these nodes in turn create new goals. The processor that was executing the parent node picks up one of the goals, allocates a new node for it (with space for the local variables of the matching clause), and starts executing it. Processors that pick up other or-parallel subgoals need only allocate a new node, since the numbering of the unbound variables is done only once. Allocating a node is a constant time operation because its size is known at compile-time.

Variable Access. When a node attempts to bind a conditional variable at address a to value v, a pair $< a, v >$ is stored in the binding list of the node, and a pointer to the value v is stored in the binding array at the position i indicated at address a. Thus binding a variable to a value is a constant-time operation. Accessing the value of a bound conditional variable is also a constant-time operation. Two accesses are required: first, to fetch the index i stored in the conditional variable, and then to fetch the value stored in the binding array at index i.

Task-Switching. When a processor switches from one leaf node n_1 to an unexplored leaf node n_2, it must construct a new binding array. It does this by making unbound, in its original binding array, all conditional variables that lie from the common ancestor of these two leaf nodes to n_1, and adds on all bindings in the binding lists of the nodes from the common ancestor to n_2. Thus, task-switching is not constant-time, and it is desirable not to switch to a very distant node in the tree. Such a scheme was recently proposed by D.H.D. Warren [W87]. He also suggests a processor scheduling policy that minimizes the number of task switches. Thus, idle processors keep moving around the or-parallel tree looking for work. When they find an unexplored branch they pick it up; no other node is explored until the entire sub-tree rooted at the current node is explored.

The binding-arrays method is attractive because it performs the most frequently occurring operation, viz., variable access, very efficiently (constant-time). The next most frequent operation, task creation, is also done efficiently (constant-time). Although task-switching is also not constant-time, its cost can be minimized by switching less frequently or switching to places that are "nearer" in the goal tree. Other properties of this method are that, if there is only one processor available, a

depth-first search would perform comparably to a sequential implemeneation, and it supports standard sequential optimizations.

3. And Parallelism

We now describe the problems of realizing and-parallelism, state desirable criteria to be satisfied by an ideal solution, and then describe the restricted and-parallelism method in some detail, as we will use it as a basis for our combined and-or implementation, described in section 4.

First, note that or-parallelism will not arise if at most one clause matches a goal at each step—such computations are said to be deterministic. Many problems, however, do lend themselves naturally to a deterministic formulation. Parallelism in such formulations comes in two common forms: divide-and-conquer parallelism and producer-consumer parallelism. In this report, we shall be concerned mainly with and-parallelism arising from divide-and-conquer formulations. In logic programs, this form of and-parallelism arises when multiple subgoals are independent of one another.

We illustrate and-parallelism in logic programs with a simple example. Consider the two clauses for the quick-sort algorithm.

```
qsort([], []).
qsort([P|L], Sorted) :-
    partition(P, L, Left, Right),
    qsort(Left, S1),
    qsort(Right, S2),
    append(S1, [P|S2], Sorted).
```

Here, the two subgoals `qsort(Left, S1)` and `qsort(Right, S2)` can be executed in and-parallel fashion, so that the two partitioned sublists of the input list are sorted simultaneously. Because the algorithm is recursive, and-parallelism similarly occurs at each recursive step. As in or-parallelism, a *task* is created for each and-parallel subgoal.

3.1 Criteria for And-parallel Implementations

Note that it is not advantageous to execute the `partition` subgoal in parallel with the two `qsort` subgoals, because the latter two depend on the former for their

input data. Similarly, the append subgoal depends upon its preceding two qsort subgoals for its input, and is best executed after they complete. While it is in theory possible to execute such "dependent" subgoals in parallel, doing so frequently results in considerable wasteful computation. In the above example, if the two qsort goals were attempted in parallel with the partition subgoal, they would end up exploring an infinite goal tree (because their first argument is unbound). Restricting their attention to a given list in their first argument narrows the search space down drastically. Thus, our first criteria is that an and-parallel implementation should avoid wasteful computation.

How we can determine which goals are independent of one another? Three different approaches to this problem have been proposed: (1) by requiring explicit *annotations* from the programmer indicating which are "input" variables and which are "output" variables [CG86]; (2) by monitoring the status for variables (bound or unbound), and dynamically re-structuring tasks to obtain optimal and-parallelism [C85]; and (3) by monitoring the status for terms (ground or nonground) and using a static task-structure, conditioned upon the status of terms, to obtain less-than-optimal (i.e., restricted) and-parallelism [D84]. Approach (1) differs from (2) and (3) in that the programmer has to explicitly specify the dependencies, using annotations. We do not further consider this approach here, because we are interested in automatic detection of and-parallelism. While a naive approach would traverse arguments of subgoals to determine if they are ground or not, clearly a desirable solution is one that avoids such a time-consuming run-time analysis. Thus, the time taken for detecting subgoal independence should be independent of the size of their respective arguments.

Unlike or-parallel implementations, a pure and-parallel implementation must be able to backtrack upon failure. To understand the problem, consider the subgoals shown below, where ';' is used between sequential subgoals—because of data dependencies—and '||' for parallel subgoals (no data dependencies).

a; b; (c || d || e); g; h

Assume that all subgoals can unify with more than one rule. A number of cases arise depending upon which subgoal fails. If subgoal a or b fails, sequential backtracking

occurs, as usual. Because c, d, and e are mutually independent, if either one of them fails, backtracking must proceed to b—but see further below. If g fails, backtracking must proceed to the right-most choice point within the parallel subgoals c || d || e, and re-compute all goals to the right of this choice point. If e were the rightmost choice point and e should subsequently fail, backtracking would proceed to d, and, if necessary, to c. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right". If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left". This latter behavior is a form of "intelligent" backtracking.

To sum up, the following criteria should be satisfied by an ideal and-parallel implementation:

1. avoid wasteful over-computation;
2. avoid complex run-time dependency analysis; and
3. support intelligent backtracking.

As with or-parallel implementations, it is desirable that an and-parallel implementation perform comparably with a sequential implementation in the single-processor case and support standard sequential optimizations.

3.2 The Restricted And-Parallel method

The following methods for and-parallel execution of logic programs have been proposed.

1. Conery's abstract parallel implementation model [C85];
2. Improvements of Conery's and-parallel model [LKL86, WC86]; and
3. Restricted And-Parallel model, introduced by DeGroot [D84], and further refined by Hermenegildo and Nasr [HN86].

Of these, the last method comes closest to realizing the criteria mentioned in the previous subsection; hence, we discuss it in some detail below.

Program representation. Program clauses are compiled into Conditional Graph Expressions (CGE). A CGE is of the form

$$(condition, goal_1, goal_2, \ldots, goal_n),$$

meaning that, if *condition* is true, goals $goal_1$ ... $goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can be either $ground(v_1, \ldots, v_n)$, which checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms or $independent(v_1, \ldots, v_n)$, which checks whether the set of variables reachable from each of v_1 ... v_n are mutually exclusive of one another. Checking for *ground* and *independence* involve very simple runtime tests, details of which are presented in [D84]. Essentially, a type tag (ground, nonground, or variable) is maintained with each term, and unification synthesizes new type tags from existing type tags. The method is conservative in that it may type a term as nonground even when it is ground—another reason why the method is regarded as "restricted". For example, the clause

```
f(X,Y) :- p(X), q(Y), s(X,Y), t(Y)
```

might be compiled into:

```
f(X,Y) :- (ground(X, Y),
                  (independent(X, Y), p(X), q(Y)),
                  (ground(Y), s(X, Y), t(Y))).
```

And-parallel execution. During forward execution, a Choice-point Marker (CM) is placed at each choice point—a choice point is created for only a sequential goal—and a Parallel-call Marker (PM) at each CGE that evaluates to true, i.e., each CGE that can actually be executed in parallel. Each PM is marked as "inside" when it is created, and the parallel resolution of the CGE subgoals is triggered. Finally, the PM mode is changed to "outside" when all subgoals report success.

When failure occurs, the most recently created marker (PM or CM) is found. If the marker is a CM, sequential backtracking occurs. If the marker is a PM and its value is "inside", all goals inside the CGE are killed, and backtracking is recursively performed. If it is a PM and its value is "outside", backtracking occurs within the CGE, right to left, until another solution is found. If no subgoal is found to succeed in this manner, failure propagates outside the CGE.

This model has an efficient implementation, because it can take advantage of WAM compiler technology to achieve standard sequential implementation optimizations, and can also efficiently accomplish a limited form of intelligent backtracking.

Further details of this approach may be obtained from [HN86].

4. Combined And-Or Parallelism

The obvious reason for combining and-parallelism and or-parallelism in a single framework is that any implementation that caters to either alone is suboptimal compared with one that caters to both. But there are other benefits too, as we will describe in the next subsection. Our approach is to combine the binding-arrays model for or-parallelism and the RAP model for and-parallelism. Thus programs are compiled into CGEs before execution. Before we present our design, we first briefly describe the main problems to be solved and then state desirable criteria.

4.1 Criteria for And-Or Parallel Implementations

Because a given logic program tends to exhibit predominately one form of parallelism, the combined model should perform as well as the pure models in these cases. Hence we adopt the union of the criteria for pure or-parallel and pure and-parallel implementations: constant variable-access, task-creation and task-switch times (pure or-parallel case); and avoidance of wasteful computation and efficient determination of subgoal independence (pure and-parallel case). Note that a combined model does not have to support any backtracking, unlike a pure and-parallel model, because of the presence of or-parallelism. The realization of and-parallelism is simplified in this respect; it suffices to detect subgoal independence and initiate their forward execution.

When there is potential for both and- and or-parallelism in a single program, exploiting either form of parallelism alone can lead to unnecessary over-computation. For example, assuming the usual definition for the `append` predicate, the pair of goals

```
? append(X, Y, [1,...,m]), append(P, Q, [1,...,n])
```

leads to a `m*n` computational cost under a pure or-parallel or pure and-parallel implementation (and also a sequential implementation), because all `n` solutions for `P` and `Q` are re-computed for each of the `m` solutions for `X` and `Y`. Since these two goals are independent, it should theoretically be possible to execute them only once, and somehow represent the cross product of their solutions. This way the

computational cost would only be of the order `m + n`. To achieve this result, the goal tree of the pure models clearly needs to be generalized to an *and-or graph*, so that sharing can be represented. For the above example, the successful leaf nodes in the or-parallel trees for the two subgoals `append(X, Y, [1,...,m])` and `append(P, Q, [1,...,n])` should be linked together to reflect the construction of the cross-product of solutions. In general, given a set of k and-parallel goals, g_1, ..., g_k, each of which having n_1, ..., n_k solutions respectively, it is desirable to achieve computational time and space much less than $n_1 * ... * n_k$.

When dealing with both and- and or-parallelism, the binding-arrays method for the pure or-parallel case must be extended to achieve constant-time access to variables. To see the problem, consider the goals (p ; (q1 || q2); r), where q1 and q2 also exhibit or-parallelism, and suppose that goal p has been completed. In order execute goals q1 and q2 in and-parallel, it is necessary to maintain separate binding arrays for them. As a result, the binding-array offsets for any conditional variables that come into existence within these two goals will overlap. Thus, when goal r is attempted, we are faced with problem of merging the binding-arrays for q1 and q2 into one composite binding-array or maintaining fragmented binding-arrays.

Finally, we should expect an and-or parallel implementation to produce solutions at least as fast as (if not much faster than) a sequential implementation. This implies that preference should be given to and-parallel tasks over or-parallel tasks if there are more tasks than available processors.

To sum up, the criteria for a combined and-or parallel implementation are essentially the union of the criteria for pure or-parallel and pure and-parallel implementations. In addition, it is desirable to avoid over-computation when both and-parallelism and or-parallelism arise within a set of goals, and also favor and-parallelism over or-parallelism if there are limited processors.

4.2 Combined And-Or Implementation

We first describe the basic structure and construction of the and-or graph, then describe how the binding-arrays method can be extended to provide constant variable-access time, and finally consider task creation and switching.

4.2.1 And-Or Graph Representation

As a motivation for our proposed representation, consider a set of k and-parallel goals, g_1, ..., g_k, each having n_1, ..., n_k solutions respectively in their *pure or-parallel tree*—the more general case is described in the next paragraph. The solution leaves in the or-parallel tree of each *adjacent* pair of and-parallel goals are linked together using $n_i * n_j$ directed links. (Note that the total number of links is $O(k * n^2)$ rather than $O(n^k)$, assuming all n_i are equal.) The structure describing the result of these k and-parallel goals is now a graph, rooted at an and-node. The solution leaves of g_1 are referred to as *solution-origin nodes* and the solution leaves of g_k are referred to as *solution-end nodes.*

To see the more general case, suppose that the above k and-parallel goals occurred as the body of some clause C, and that clause C was invoked by goal a_i, which occurs amidst and-parallel goals $(a_1 \parallel \ldots \parallel a_n)$. Suppose further that the and-or graphs of each a_i have been constructed, and we now wish to construct the graph for the result. All we need to do is to link the solution-end nodes of a_{i-1} with the solution-origin nodes of a_i, and the solution-end nodes of a_i with the solution-origin nodes of a_{i+1}. The solution-origin nodes of the result are those of a_1; similarly, the solution-end nodes of the result are those of a_n.

To combine the solutions of k or-parallel goals, we take the disjoint union of their respective solution-origin and solution-end nodes. To execute a goal g_{k+1} sequentially after a goal g_k, we root its and-or graph below each solution-end node of g_k.

Example. Figure 2 illustrates this construction for a simple example—we do not consider how variable bindings are represented here; this is discussed in the next subsection. Note that there are two kinds of nodes, and-nodes (bold-face) and or-nodes, and three kinds of directed edges, and-arcs (bold-face), or-arcs, and solution-links (curved). The top-level node is an or-node. Associated with each node is a goal-list. All and-nodes have just a single subgoal—the one for which the node was created. The goal-list for an or-node consists of any remaining subgoals of its parent appended to any subgoals in the body of the matching clause that created the or-node.

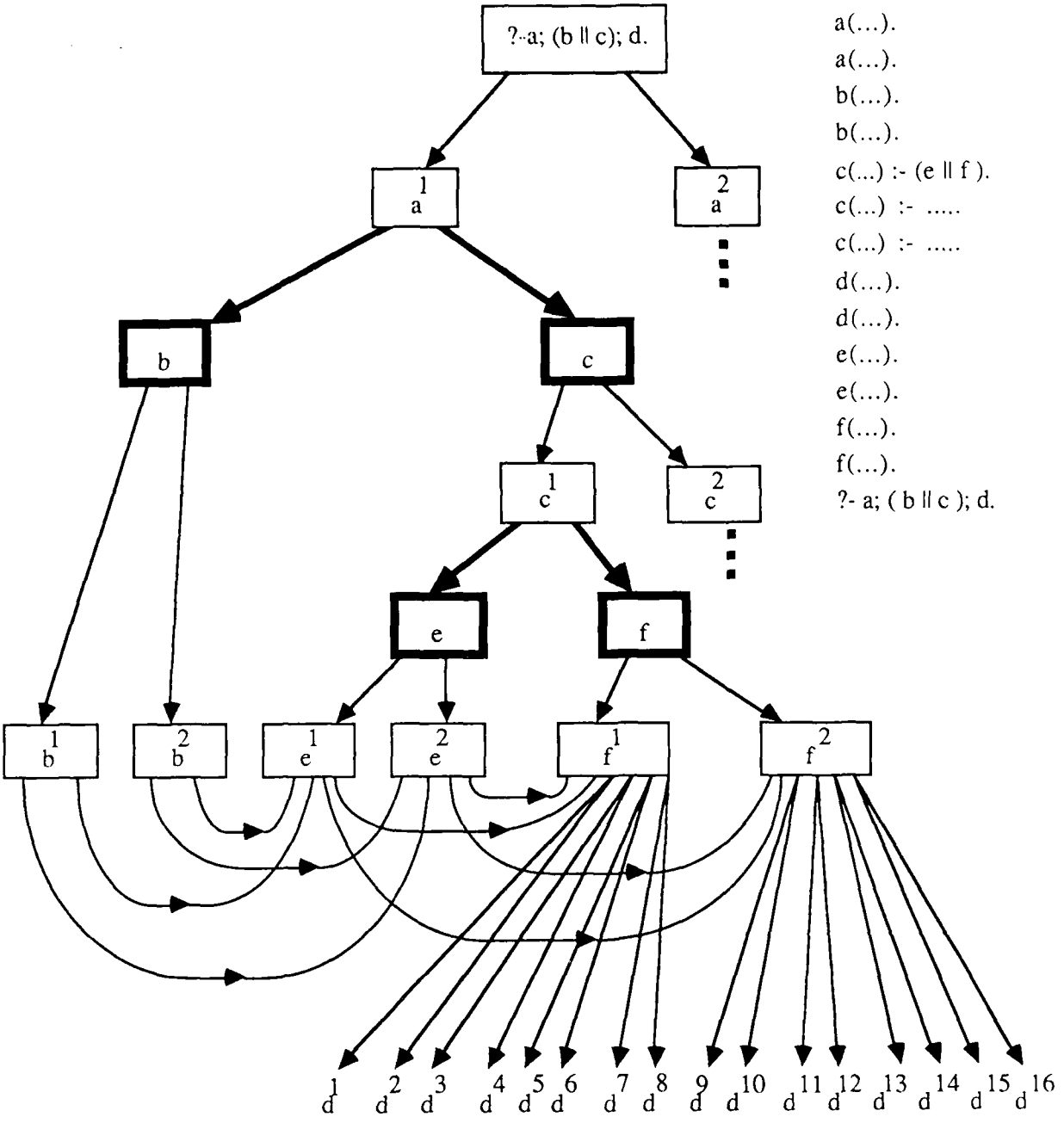


Figure 2. An Example of an And-Or DAG

We briefly explain below how the graph of figure 2 would have been constructed. First, because subgoal 'a' in the top-level node cannot be executed in and-parallel with any other subgoal, it is initially explored in pure or-parallel fashion. Assuming it unifies with both clauses for 'a', the goal-lists of the or-nodes labelled a^1 and a^2 are just the remaining subgoals in their parent node, because the clauses for 'a' are unit clauses. The or-node a^1 then executes the subgoals b and c in and-parallel by creating and-nodes for them, initialized with the goals b and c respectively. Execution similarly continues at and-nodes b and c until the nodes b^1, b^2, e^1, e^2, f^1 and f^2 are finally created.

The solution-links shown in figure 2 represent the connections between solution-end nodes and solution-origin nodes. For example, the solution-end nodes of the and-node b are b^1 and b^2, and these are linked to the solution-origin nodes of c shown in the figure, namely e^1 and e^2. Finally, because goal d is to be executed sequentially after the goals (b || c), we root its and-or graph below each solution end-node of c, namely, f^1 and f^2. Because goal d could depend upon both goals b and c for the bindings of its variables, it is necessary to construct as many and-or graphs for d as there are solutions in the cross-product of b and c.

We defer until section 4.2.3 the details of how concurrently executing processors perform solution-linking. The reader may verify that there is one-to-one correspondence between each path in an or-parallel tree and each path in the corresponding and-or graph starting from the root node, proceeding via solution-origin and solution-end nodes, and ending on a solution-end node of the root.

4.2.2 Binding Arrays

We now explain our proposed extension to the binding-arrays method in order to accomodate and-parallelism.

Suppose that an or-node is about to create two or more and-nodes and there are processors available to execute them in and-parallel. For each such and-node, the assignment of offsets is re-started from 0 by simply resetting the counter associated with the and-node. This resetting is not needed for the left-most and-node because it could be picked up by the processor executing the parent or-node. We will refer to and-nodes where the offsets are reset to 0 as *offset-origin* nodes. For each such

node, the pair $< a, n >$ is stored in a *cache* local to the processor, where a is the address of the offset-origin node and n is the index of the next free location in the processor's binding array. Every node records the address, a, of its offset-origin and-node.

When a reference to a conditional variable v occurs, we calculate its offset in the binding array in two steps: first, we present the address a of the offset-origin and-node for v to the cache and obtain the value n; then we access the binding array at offset $(n + i)$, where i is the offset stored with v. Note that access to variables is still constant-time, though the constant is somewhat larger compared to the binding-arrays method for pure or-parallelism.

4.2.3 Task Creation and Switching

There are two sources of work for the processors: unexplored or-nodes and unexplored and-nodes. If there are insufficient processors to explore all nodes, and-nodes are given preference over or-nodes. Thus, if processors p_1 and p_2 are exploring respectively two sibling and-nodes (f || g), and p_1 has completed one or-parallel path arising from f, it will not explore other or-parallel paths rooted at f until all other and-parallel goals have been explored. If there are insufficient processors to explore all and-nodes, only one of the many possible or-parallel paths will be explored at any and-node. Thus, if there is one processor, p_1, available to execute the and-parallel goals (f || g) and p_1 has completed one or-parallel path arising from f, it will explore an or-parallel path rooted at g before returning to other or-parallel paths in f.

Task-creation is a constant-time operation because it is performed identical to the pure or-parallel model. Task-switching to a node on a different or-parallel path is also identical to the pure or-parallel model (and is not constant-time). Task-switching to a node on a different and-parallel path requires more work, as solutions must be linked. Each and-node therefore maintains a set of solution-origin addresses and a set of solution-end addresses. A leaf node's address is included in the set of solution-origin addresses of the closest ancestor and-node (*solution-origin owner*) that has a left sibling; similarly, the leaf node's address is included in the set of solution-end addresses of the closest ancestor and-node (*solution-end owner*) that

has a right sibling. For example, in figure 2, leaf node e^2 will be included in the set of solution-origin addresses maintained by c and the set of solution-end addresses maintained by e.

When a processor finishes executing an or-parallel path it adds the address of the leaf node into the appropriate solution-origin and solution-end sets. If the leaf node is not a solution-origin node of the entire goal tree, the processor links to it each node referred to in the set of solution-end addresses of the left sibling of its solution-origin owner. Similarly, if the leaf node is not a solution-end node of the entire goal tree, the processor links it to each node referred to in the set of solution-origin addresses of the right sibling of its solution-end owner.

When a processor completes execution at some leaf node and looks for more work, it should delete the $< a, n >$ pair from its cache for each offset-origin and-node a that is no longer applicable. Also, when a processor located at a solution-end node e links to a solution-origin node o, it loads its binding array from the binding lists of all nodes in the path from node e up to the common ancestor of o and e. During this process, the processor also records the $< a, n >$ pairs in its cache for all offset-origin nodes found. Clearly this is not constant-time, but this is in lieu of having to re-compute the possibly multiple solutions of the sibling and-node. Thus, in the example in figure 2, when a processor links the solution-end node b^1 to the solution-origin node e^1 it will have to make an entry for nodes c and e into its cache. This is to ensure that descendent nodes of f^1 or f^2 can access variables in c.

5. Conclusions and Related Work

The combined and-or model presented in this paper preserves the characteristics of the binding-arrays method for pure or-parallelism and the RAP method for pure and-parallelism, namely, constant-time variable access, constant-time task-creation, efficient dependency checking of subgoals, and restricted intelligent backtracking. Additionally, there is no restarting of and-parallel goals required as in the RAP, and the computation of and-parallel subgoals are shared across different solution paths, resulting in better time and space performance. Standard optimizations, such as last-call and environment trimming, still apply, though the conditions under which they can be applied would slightly change due to the sharing of nodes. Fur-

thermore, if there is only one processor available, the execution would be as efficient as a sequential implementation, with the added advantages of limited intelligent backtracking, no redundant computations and no restarting of and-parallel goals. The one main shortcoming of this approach is the high cost of task switching—a shortcoming inherited from the binding-arrays method—but we propose to keep the task granularity high, so as to keep this overhead at a minimum.

To the best our knowledge, very few research projects have aimed at realizing both and- and or-parallelism in a single implementation:

1. Conery's And-Or Process Model [CK81].
2. PEPSys model from ECRC [WR87].
3. Wise's Epilog [W86].

Conery's and-or model generates an or-process for each clause matched and an and-process for each subgoal which can be executed in and-parallel. The model constructs a run-time data-flow graph to detect potentially independent subgoals. While the model detects maximal parallelism, its overheads are high because of the need to reconstruct the data-flow graph after every successful subgoal and the need to pass answer-substitutions back and forth by message-passing. Also, there could be unnecessry over-computation when both and-parallelism and or-parallelism exist in a set of goals.

The PEPSys model uses the technique of time-stamping to dereference the correct value of a variable. Locating a conditional variable appears to be a non-constant-time operation. The model requires the user to annotate programs with special operators to exploit and-parallelism. A *join* is used to obtain all the solutions of two and-parallel subgoals. This approach could be wasteful because not all joins may be necessary to report a top-level solution. This also requires synchronization when bindings of one and-parallel subgoal has been computed before the other. We think that traversing different paths to obtain different solutions is a more efficient approach than forming all possible joins. The model may result in over-computation, like Conery's model, when both and-parallelism and or-parallelism exist in a set of goals. The advantage of the PEPSys model is, however, that it is not tied down to a fixed architecture and can be implemented on shared or

non-shared memory multiprocessors.

Wise's Epilog is based upon or-parallelism and unrestricted and-parallelism. As explained earlier, unrestricted and-parallelism could often leading to wasteful computation; besides, the *back-unification* necessary to maintain consistency is an extra overhead. Wise's model may also over-compute when both and-parallelism and or-parallelism exist in a set of goals.

We are in the process of further refining our high-level definition of the combined and-or parallel implementation, and expect to develop a simulator of this model before constructing an actual parallel implementation.

References

[B84] P. Borgwardt, "Parallel Prolog using stack segments on shared memory multiprocessors", In *Proceedings of the 1984 International Sumposiuom on Logic Programming,*", Atlantic City, NJ, 1984. pp. 2-11.

[BL86] R. Butler, E. L. Lusk, R. Olson and R. Overbeek, "ANLWAM: A Parallel Implementation of the Warren Abstract Machine", Internal Report, Argonne National Laboratory, U.S.A., 1986.

[CG86] K. Clark and S. Gregory, "Parlog: Parallel Programming in Logic", In *A.C.M. TOPLAS*, Vol. 8, No. 1, Jan. 1986.

[CH86] A. Ciepielewski and B. Hausman, "Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs", In *1986 IEEE Symposium on Logic Programming*, Salt Lake City. pp. 246-257.

[C87] J. S. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors", *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 457-467

[CK81] J. S. Conery and D. F. Kibler. "Parallel Interpretation of Logic Programs", *Proceedings of the Conference on Functional Languages and Computer Architecturea*, Oct., 1981. pp. 163-170.

[CK83] J. S. Conery and D. F. Kibler. "And Parallelism in Logic Programs", In *Proceedings of the International Joint Conference in AI*, 1983.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[CY86] S Chengzheng and T. Yungui, "The Or-forest Description for the Execution of Logic Programs", *3rd International Conference on Logic Programming*, London, 1986. pp. 710-717.

[D84] D. DeGroot. "Restricted AND-parallelism", *Int'l Conf. on Fifth Generation Computer Systems*, Nov., 1984.

[D87] D. DeGroot. "Restricted AND-parallelism and Side Effects", in *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 80-88.

[DLO87] T. Disz, E. Lusk, and R. Overbeek, "Experiments with OR-parallel Logic Programs", In *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[HCH87] B. Hausman, A. Ciepielewski, and S. Haridi, "Or-Parallel Prolog made efficient on shared memory multiprocessors", in *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[H86] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd International Conference on Logic Programming*, London, 1986.

[HN86] M. V. Hermenegildo and R. I. Nasr, "Efficient Implementation of backtracking in AND-parallelism", *3rd International Conference on Logic Programming*, London, 1986. pp. 40-54.

[JG86] B. Jayaraman and G. Gupta, "Parallel Execution of an Equational Language" *In Proceedings of the Workshop on Graph Reduction*, Santa Fe, 1986, Lecture Notes in Computer Science, Vol 279, pp 370-381.

[K74] R. A. Kowalski. "Predicate Logic as a Programming Language", *Proc. IFIPS* 1974.

[L84] G. Lindstrom, "Or-Parallelism on Applicative Architectures", in *Sec-*

ond International Logic Programming Conference", Uppsala, Sweden. 1984.

[LKLS6] Y. Lin, V. Kumar, and C. Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs", In *3rd International Conference on Logic Programming*, London, 1986.

[SWS7] K. Shen and D. H. D. Warren, "A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog", In *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[S83] E. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", ICOT Tech. Report TR-003, ICOT, Tokyo, Feb., 1983.

[TL87] P. Tinker and G. Lindstrom, "A Performance Oriented Design for Or-parallel Logic Programming", *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[W83] D. H. D. Warren, "An Abstract Instruction Set for Prolog". Tech. Note 309, SRI International, 1983.

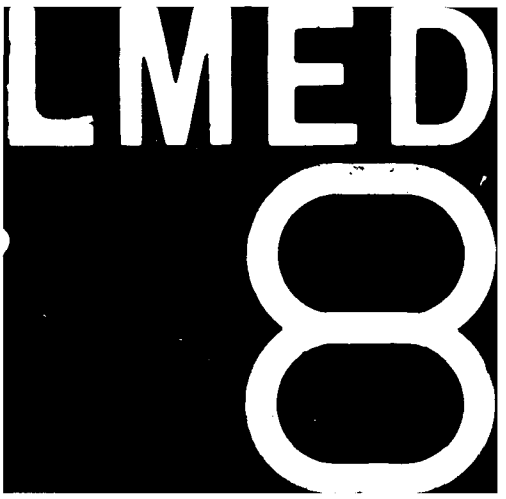
[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog - Abstract Design and Implementation Issues", *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[W87a] D. H. D. Warren, "Or-Parallel Execution Models of Prolog". TAPSOFT '87, Springer Verlag, LNCS 250.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies", In *The 1984 International Symposium on Logic Programming*, Atlantic City, pp. 198-202.

[WR87] H. Westphal, P. Robert, J. Chassin and J. Syre, "The PEPSys Model: Combining Backtracking, AnD- and OR-parallelism". In *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 436-448.

[W86] D. S. Wise, "Prolog Multiprocessors", Prentice-Hall, 1986.


LMED
8